Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2011) 1–9
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Block-asynchronous Multigrid Smoothers for GPU-accelerated Systems

Hartwig Anzt[a,1,*], Stanimire Tomov[b,*], Mark Gates[b,*], Jack Dongarra[b,*], Vincent Heuveline[a,*]

[a] *Karlsruhe Institute of Technology (KIT) Germany*
[b] *University of Tennessee, Knoxville, USA*

Abstract

This paper explores the need for asynchronous iteration algorithms as smoothers in multigrid methods. The hardware target for the new algorithms is top-of-the-line, highly parallel hybrid architectures – multicore-based systems enhanced with GPGPUs. These architectures are the most likely candidates for future high-end supercomputers. To pave the road for their efficient use, we must resolve challenges related to the fact that data movement, not floating-point operations, is the bottleneck to performance. Our work is in this direction — we designed block-asynchronous multigrid smoothers that perform more flops in order to reduce synchronization, and hence data movement. We show that the extra flops are done for "free," while synchronization is reduced and the convergence properties of multigrid with classical smoothers like Gauss-Seidel can be preserved.

Keywords: Block-asynchronous Iteration, Multigrid Smoothers, GPU

1. Introduction

Classical relaxation methods such as Gauss-Seidel and Jacobi require a synchronization between each iteration, which implies a severe restriction for parallel implementations. An asynchronous iteration method removes this synchronization barrier, updating components using the latest available values. It allows a large freedom in the update order and the number of updates per component, while every component update uses the latest available values for the other components. In the end, the obtained algorithm is neither deterministic nor does it imply convergence for all systems that can be solved by the classical Jacobi approach. In fact, it requires the linear equation system to fulfill additional conditions. While due to the restrictive convergence properties, an asynchronous iteration method may seem to be unattractive from a mathematical point of view, in contrast to most other iterative methods, it is able to exploit the high computational power of modern hardware platforms. The high parallelization potential and the high tolerance to communication and synchronization latencies make it a perfect candidate for computing systems that consist of a large number of parallel computing cores that provide excellent performance as long as they run

*

Email addresses: hartwig.anzt@kit.edu (Hartwig Anzt), tomov@eecs.utk.edu (Stanimire Tomov), mgates3@eecs.utk.edu (Mark Gates), dongarra@eecs.utk.edu (Jack Dongarra), vincent.heuveline@kit.edu (Vincent Heuveline)

[1] Corresponding author

independently, but suffer severely from data transfers and synchronization points. For linear equation systems fulfilling the required additional conditions, an asynchronous iteration is often able to more than compensate for the inferior convergence rate by leveraging the computational power of the available hardware resources. This leads to the fact that, for many problems, an asynchronous iteration outperforms the classical synchronized relaxation methods like Gauss-Seidel and Jacobi [1].

While iterative methods based on matrix splitting are nowadays seldom applied directly to solve a problem, they are often used in multigrid methods, to smooth the error terms related to the large eigenvalues on the distinct grid levels [2, 3]. The superiority of an asynchronous iteration method has been shown for many linear equation problems, but it is an open question whether it is suitable to replace the classical smoothers in multigrid methods. Since the parallelization of smoothers is usually crucial when optimizing multigrid solvers [4], it may have a huge impact on the overall multigrid performance.

Targeting this topic we split this paper into different parts: We start with an introductory section providing the mathematical background of geometric multigrid methods and asynchronous iteration. In the second part we give details about the implementation we used for the numerical tests and the hardware configuration. We then report numerical results of multigrid implementations using asynchronous iteration smoothers and compare them to Gauss-Seidel schemes. In the last section we conclude and provide ideas about which topics could be interesting to address in this research field.

2. Mathematical Background

2.1. Multigrid Methods

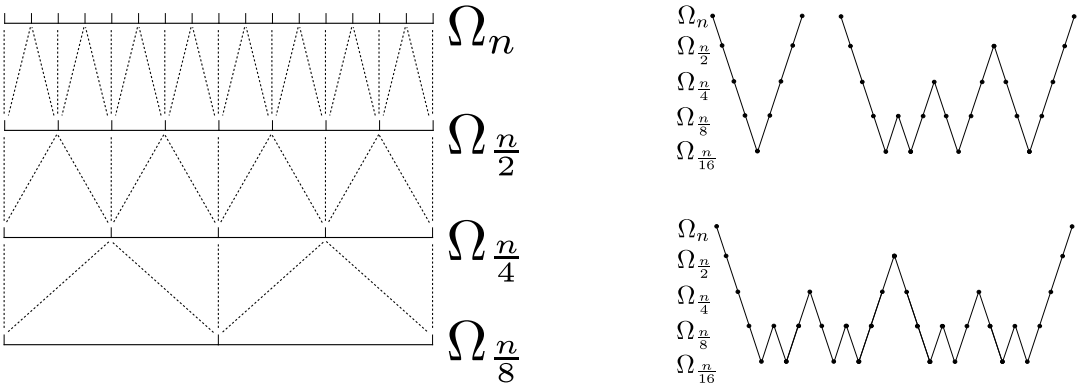
Multigrid methods are a type of error correction method that attempts to find a solution approximation by using a sequence of problems that are similar with respect to their structure, but differ in having successively decreasing dimension [3, 5, 6]. They are usually applied to problems occurring in the field of finite element or finite difference discretizations of partial differential equations. In this case, different discretizations with decreasing dimension of the continuous problem can be used for the sequence of discretized problems, as in Figure 1a. This enables splitting the approximation error into high and low frequency terms that can then be treated with different efficiency on the distinct grid levels. A basic multigrid algorithm is given in Algorithm 1.

Using an optimal combination of problem sequence and operators, one can obtain a solver with optimal complexity $O(n)$. Hence, multigrid solvers are among the most efficient solvers for the discretized partial differential equations.

```
MG(x_l, b_l, l)
if l = 0 then
    Solve A_l x_l = b_l {exact solution on coarsest grid}
else
    x_l = S_l^{ν_1}(x_l, b_l) {pre-smoothing}
    r_{l-1} = r_l^{l-1}(b_l − A_l x_l) {restriction}
    v_{l-1} = 0
    for j = 0; j < γ; j++ do
        MG(v_{l-1}, r_{l-1}, l − 1) {coarse grid correction}
    end for
    x_l = x_l + p_{l-1}^l v_{l-1} {prolongation of coarse grid correction}
    x_l = S_l^{ν_2}(x_l, b_l) {post smoothing}
end if
```

Algorithm 1: Basic multigrid method [3].

In the case of using only two grids and solving the error correction equation exact on the grid $\Omega_{\frac{n}{2}}$, the method is called the "Two-Grid Iteration Method." Usually, the process is recursively applied to successively coarser grids, creating a "Multigrid Method." In this case there exist different schemes for how to organize the multigrid process. The structure is determined by the number R of error correction computations on every grid level. Implementations usually choose $R = 1$ or $R = 2$. The resulting multigrid iteration schemes are called "V-cycle" and "W-cycle,"



(a) Sequence of successively coarser grids.

(b) Visualizing V-cycle, W-cycle and F-cycle.

Figure 1: Basic Multigrid Principles.

respectively. While the V-cycle is usually very efficient in terms of computational cost, it can be unstable with respect to the properties of the problem. The W-cycle is computationally more expensive, but at the same time more robust in terms of the problem. For $R \geq 3$, the multigrid iteration method becomes inefficient. A trade off between V-cycle and W-cycle is the "F-cycle," shown in Figure 1b.

2.2. Smoother in Multigrid Methods

One critical component of multigrid methods is the smoother. Usually, a simple relaxation method such as Gauss-Seidel or Jacobi is used for pre- and post-smoothing the solution approximations on the distinct grid levels. The idea of applying the smoother is to make the underlying error smooth so that it can be approximated efficiently on a coarser grid. From the analytical point of view, if the error is expressed in terms of the eigenvectors of the system, the smoother must eliminate the error associated with the eigenvectors having large eigenvalues, while the coarse-grid correction eliminates the remaining error associated with eigenvectors having small eigenvalues [3].

The best smoothers, such as Gauss-Seidel, usually do not parallelize well. Therefore, much effort is put into developing parallel smoothers that scale on multicore architectures. The main approach is to use a set of local smoothers that exchange boundary values in a Jacobi-like manner [7, 8]. The performance of these hybrid smoothers may then be enhanced furthermore by using weights [9].

Still, the synchronization necessary to exchange boundary values may be detrimental to the performance on highly parallel architectures. Therefore, it is worthwhile to consider a block-asynchronous iteration, which lacks any synchronization and therefore scales optimally on any architecture, for the smoother in multigrid methods.

2.3. Asynchronous Iteration

The motivation for an asynchronous iteration is modern hardware, which provides a large number of cores that achieve excellent performance when running in parallel, but suffer when synchronizing or exchanging data. Therefore, algorithms that lack any synchronization achieve outstanding performance on these devices, while most of the relaxation methods are poorly parallel and require regular data exchange. For computing the next iteration in relaxation methods, one usually requires the latest values of all components. For some algorithms, e.g., Gauss-Seidel [10], even the already computed values of the current iteration step are used. This requires a strict order of the component updates, limiting the parallelization potential to a stage, where a component cannot be updated again until all the other components are updated.

If this order is not adhered to, i.e., the individual components are updated independently and without consideration of the current state of the other components, the resulting algorithm is called a chaotic or asynchronous iteration method. In the 1970s, Chazan and Miranker analyzed some basic properties of these methods and established convergence theory [11, 12, 13, 14, 15]. For the last 30 years, these algorithms went out of favor due to the superior convergence properties of synchronized iteration methods like the Krylov subspace methods. Today, due to the complexity of heterogeneous hardware platforms and the large number of computing units in parallel devices like GPUs,

these schemes may become interesting again for applications like multigrid methods, where highly parallel smoothers are required on the distinct grid levels. While traditional smoothers like the sequential Gauss-Seidel obtain their efficiency from their fast convergence, an asynchronous iteration scheme may compensate for its inferior convergence behavior by superior scalability.

The chaotic or asynchronous relaxation scheme defined by Chazan and Miranker [11] can be characterized by two functions, an update function $u(\cdot)$ and a shift function $s(\cdot,\cdot)$. For each non-negative integer ν, the component of the solution approximation x that is updated at step ν is given by $u(\nu)$. For the update at step ν, the mth component used in this step is $s(\nu, m)$ steps back. All the other components are kept the same. This can be expressed as:

$$x_l^{\nu+1} = \begin{cases} \sum_{m=1}^{N} b_{l,m} x_m^{\nu-s(\nu,m)} + d_l & \text{if } l = u(\nu) \\ x_l^{\nu} & \text{if } l \neq u(\nu). \end{cases} \tag{1}$$

Furthermore, the following conditions can be defined to guarantee the well-posedness of the algorithm [16]:

1. The update function $u(\cdot)$ takes each of the values l for $1 \leq l \leq N$ infinitely often.
2. The shift function $s(\cdot,\cdot)$ is bounded by some $\bar{s}$ such that $0 \leq s(\nu, m) \leq \bar{s}\ \forall\ \nu \in \{1, 2, \ldots\}, \forall\ m \in \{1, 2, \ldots, N\}$. For the initial step, we additionally require $s(\nu, m) \leq \nu$.
3. The shift function $s(\cdot,\cdot)$ is independent of m.

If these conditions are satisfied and $\rho(|M|) < 1$ (i.e., the spectral radius of the iteration matrix, taking the absolute values of its elements, is less than one), the convergence of the asynchronous method is guaranteed [16].

3. Numerical Experiments

3.1. Experimental Setup

The numerical problem we target is the finite difference discretization of the differential equation

$$-\Delta u + \varepsilon u = f,$$

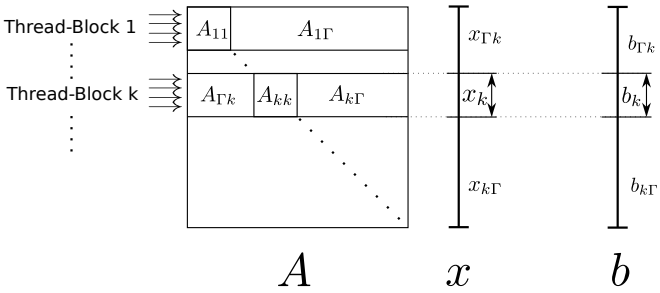
which for $\varepsilon = 0$ becomes the Laplace equation. For Dirichlet boundary condition equal to zero, the 1D discretization for this problem on a grid of size h can be written as a system of linear equations of the form $Ax = b$ where $b = h^2 f$ and

$$A = \begin{pmatrix} 2+h^2\varepsilon & -1 & 0 & \ldots & 0 \\ -1 & 2+h^2\varepsilon & \ddots & \ddots & \vdots \\ 0 & \ddots & 2+h^2\varepsilon & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \ldots & 0 & -1 & 2+h^2\varepsilon \end{pmatrix}. \tag{2}$$

Although this may seem to be a very basic problem, it contains many essential aspects necessary to analyze the convergence behavior of the multigrid method. We vary $\varepsilon \in [10^{-6},\ 10^{-1}]$ to control the condition number κ of the linear equation system. Using the Gerschgorin circle theorem [17], we can estimate the respective condition numbers by $\kappa \approx 4 \cdot \frac{1}{h^2\varepsilon}$. The first experiment in Section 3.3 will analyze the impact of this additive component determining the condition number.

The geometric multigrid method we apply to this system is implemented according to the Algorithm 1, where we use the Conjugate Gradient method for the solution of the coarse grid system. To analyze the performance of a GPU-based block-asynchronous iteration as smoother, we compare it with a CPU implementation of Gauss-Seidel performing smoothing iteration.

For all smoothers, we use a stencil implementation of the corresponding linear equation system, updating the distinct components by using the adjacent components. This reduces the computational cost, since we do not have to perform a sparse matrix vector multiplication, as well as the memory requirements, which are usually daunting when performing GPU-based kernels. Still, we utilize the explicit matrix to compute the error term on each grid level.



$$x_k + = D_{kk}^{-1} \left(b_k - A_{\Gamma k} x_{\Gamma k} - A_{kk} x_k - A_{k\Gamma} x_{k\Gamma} \right)$$

Figure 2: Visualizing the asynchronous iteration in block description used for the GPU implementation.

Utilizing stencils for this may be beneficial for the overall performance as well, but we refrain from doing so since this is not the main target of this paper.

In general, it is challenging to analyze the performance of a smoother within a multigrid framework. The reason is that the smoothers' properties depend on tunable parameters, e.g., related to the linear system characteristics, the applied multigrid scheme, the solver used on the coarsest grid, etc. We therefore split the numerical tests into two parts, where we first analyze the two-grid iteration and then extend it to a complete multilevel V-cycle.

Enhancing the asynchronous iteration method by local component updates in every thread block (see the CUDA terminology [18]) has led to the development of a block-asynchronous iteration [1]. While the motivation for this scheme is the design of graphics processing units and the CUDA programming language, it is equivalent to a two-staged asynchronous iteration [19]. We split the linear system into blocks of rows, and then assign the computations for each block to one thread block on the GPU. Between these thread blocks, an asynchronous iteration method is used, while within each thread block, multiple Jacobi-like iterations are performed, instead of a single iteration. During these local iterations, the x values used from outside the block are kept constant, equal to their values at the beginning of the global iteration. After the local iterations, the updated values are communicated. This approach is inspired by the well know hybrid relaxation schemes [4, 20]. In other words, using domain-decomposition terminology, our blocks correspond to subdomains and thus we iterate locally on every subdomain. We denote this scheme by *async-(i)*, where the index i indicates that we use i Jacobi-like updates on the subdomain. As the subdomains are relatively small and the data needed largely fits into the multiprocessor's cache, these additional iterations on the subdomains come for almost free. The obtained algorithm, visualized in Figure 2, can be written as a component-wise update of the solution approximation:

$$x_k^{(m+1)} + = \frac{1}{a_{kk}} \left(b_k - \underbrace{\sum_{j=1}^{T_S} a_{kj} x_j^{(m-\nu(m+1,j))}}_{\text{global part}} - \underbrace{\sum_{j=T_S}^{T_E} a_{kj} x_j^{(m)}}_{\text{local part}} - \underbrace{\sum_{j=T_E}^{n} a_{kj} x_j^{(m-\nu(m+1,j))}}_{\text{global part}} \right), \tag{3}$$

where T_S and T_E denote the starting and the ending indexes of the matrix/vector part in the thread block. Furthermore, for the local components, the antecedent values are always used, while for the global part, the values from the beginning of the iteration are used. The shift function $\nu(m+1, j)$ denotes the iteration shift for the component j, which can be positive or negative, depending on whether the respective other thread block has already conducted more or less iterations. Note that this gives a block Gauss-Seidel flavor to the updates. It should also be mentioned that the shift function may not be the same in different thread blocks.

We can identify three parameters that can be used to adjust the smoother. These are the number of global iterations that correspond to the number of iterations of a synchronized iterative method like Jacobi or Gauss-Seidel, the number of local iterations on the subdomains, and the size of the subdomains. The number of global iterations is usually daunting concerning the execution time of a block-asynchronous iteration on GPUs, which is still small compared to

synchronized Gauss-Seidel on the CPU. Adding local iterations on the GPU, due to the data locality and the GPU architecture, basically comes for free [1]. But at the same time, adding local iterations may not trigger the same improvement to the solution approximation. Since in general the factor between the convergence rate of Gauss-Seidel and Jacobi equals to two, we always merge two global block-asynchronous iterations into one smoothing step. The local iterations may then be used to compensate for the convergence loss due to the chaotic behavior. Without investigating the trade-off between global and local iterations, we set the latter one to the fixed number of five, and denote the obtained block-asynchronous iteration with *async-(5)*. We also want to neglect the issue of the non-deterministic behavior of *async-(5)*, and refer to all further results as average.

In the second part of the numerical experiment section we then extend the Two-Grid iteration to a full V-cycle. We analyze the impact of adding grid levels, and report the smoother run times for different problem sizes. Finally, we provide a detailed time-to-solution comparison between block-asynchronous iteration and Gauss-Seidel smoothed multigrid for a 10-level implementation using different numbers of smoothing steps.

3.2. Hardware and Software Issues

The experiments were conducted on a heterogeneous GPU-accelerated multicore system located at the Karlsruhe Institute of Technology, Germany. The system is equipped with two Intel XEON E5540 @ 2.53GHz and 4 Fermi C2070 (14 Multiprocessors x 32 CUDA cores @1.15GHz, 6 GB memory). The GPUs are connected to the host through a PCI-e×16.

On the CPU, the synchronous Gauss-Seidel implementation runs on 4 cores. Intel compiler version 11.1.069 [21] is used with optimization flag "-O3". The GPU implementation is based on CUDA [18], while the respective libraries used are from CUDA 4.0.17 [22]. The component updates were coded in CUDA, using thread blocks of size 512. The kernels are then launched through different streams. The thread block size, the number of streams, along with other parameters, were determined through empirically based tuning.

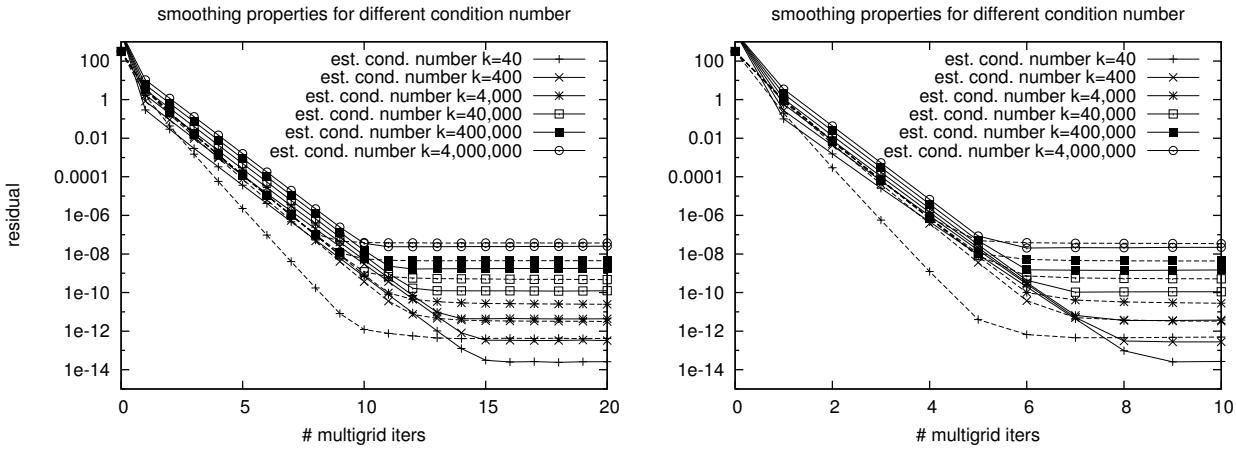
3.3. Numerical Experiments

In the first experiment, we analyze the impact of the condition number of the linear equation system on the performance of multigrid methods smoothed by block-asynchronous iteration and Gauss-Seidel, respectively. We choose a dimension of $n = 10,000,000$ and compare for different condition numbers κ the convergence with respect to the iterations. Figure 3a shows numerical results using one smoothing step of Gauss-Seidel or block-asynchronous iteration, while the results in Figure 3b are for two smoothing steps. First, we observe that the number of necessary multigrid steps to convergence can be considerably decreased by performing two instead of one smoothing iteration. Second, the block-asynchronous smoother has smoothing properties similar to the Gauss-Seidel, so the convergence behavior of the multigrid is not affected by the respective smoother. Only for very small condition numbers, the block-asynchronous iteration performs even better than the Gauss-Seidel smoother. The only difference is the accuracy of the final solution: The Gauss-Seidel method allows a higher approximation quality than the block-asynchronous iteration. But the variations are small and the more crucial factor determining the accuracy of the final solution approximation are the limitations of the floating point format. Still, if very accurate solution approximations are requested, it may also be reasonable to switch to a Gauss-Seidel method for the last V-cycles.

In the next experiment we investigate the impact of the problem size on the finest grid level. For this purpose we choose problem sizes between 10^4 and 10^8 and analyze the convergence behavior. The results shown in Figure 4 reveal that the problem size has almost no influence on the convergence rate.

While the convergence rate with respect to iteration number is interesting from the theoretical point of view, the more relevant factor is the convergence with respect to time. This depends not only on the convergence rate, but also on the efficiency of the respective algorithm on the available hardware resources. While the Gauss-Seidel smoother requires a strict update order, the block-asynchronous iteration is tolerant to varying update orders and synchronization latencies, and therefore enables a highly parallel and synchronization free GPU implementation. In Table 1, we report run times of the respective smoothers for different problem sizes. We also extend the analysis from a two-grid method to multiple levels.

The run times are the aggregated smoother times for the multigrid to converge. Since we usually obtain higher accuracy approximations for the Gauss-Seidel smoother, we choose a stopping criterion for the multigrid iteration that can be achieved for both methods. Additionally we provide the data transfer time for the GPU implementation of



(a) One Pre-/Post-Smoothing Step (b) Two Pre-/Post-Smoothing Steps

Figure 3: Two-level convergence for $n = 10,000,000$ and different condition numbers using one or two smoothing steps, respectively. Dashed lines are block-asynchronous and solid lines are Gauss-Seidel.

the block-asynchronous iteration smoother. This also contains the overhead of the GPU initialization. Note that we report only the run times for the smoother, which increase for multiple levels due to additional smoother calls. The total runtime for the multigrid iteration may still decrease with more levels due to the smaller linear equation system solved on the lowest grid level.

	2 levels			3 levels			4 levels			5 levels		
dim	G.-S.	async-(5)	transfer	G.-S.	async-(5)	transfer	G.-S.	async-(5)	transfer	G.-S.	async-(5)	transfer
10k	0.00398	0.00085	0.01193	0.00621	0.00181	0.02243	0.00805	0.00242	0.03289	0.00957	0.00338	0.04102
100k	0.03613	0.00602	0.02571	0.05539	0.00913	0.04267	0.06621	0.01044	0.05673	0.07181	0.01156	0.06866
1,000k	0.36815	0.05397	0.10779	0.54973	0.08433	0.16522	0.64393	0.09654	0.20734	0.69662	0.10531	0.23634
10,000k	3.62690	0.53082	0.85057	5.60806	0.80394	1.29404	6.48827	0.92531	1.53345	6.95943	1.00858	1.64843

Table 1: Average smoother runtime [s] for Gauss-Seidel (G.-S.) on CPU, block-asynchronous iteration (async-(5)) on GPU and the data transfer time performing 2 Pre- and 2 Post-smoothing steps on the respective grid levels.

We observe in Table 1 that for Gauss-Seidel on the CPU, linearly increasing the problem size on the finest grid corresponds to a linear run time increase for the smoother. This is also true for the block-asynchronous iteration on the GPU, except for small problem sizes, where calling the GPU kernels triggers some overhead. Also the data transfer is also influenced by the GPU initialization. For all problem sizes and grid sequences, the async-(5) smoother outperforms the Gauss-Seidel smoother. While for small problems the improvement is at least a factor of three, it rises to 7 for larger dimensions. Since the multigrid framework is often implemented on the host of the system, we should also take the data transfer time into account. Then, for small problem sizes, the async-(5) smoother suffers from this overhead due to the GPU initialization and expensive data transfer. For larger problem sizes, the async-(5) smoother outperforms the Gauss-Seidel smoother at least by a factor of two, independent of the number of grid levels.

The question is how this corresponds to an acceleration of the multigrid method, since the smoothers usually accounts for a small part in the overall execution time. To investigate this issue, we apply a 10-level multigrid method to a very ill-conditioned problem of size 10,000,000 and provide detailed analysis on the execution time of the smoother, the grid operations like restriction, prolongation and residual computation, and the direct solver on the coarsest grid level.

Analyzing the results, we realize that applying more smoothing steps reduces the number of V-cycles in the multigrid method, which again reduces the number of solver calls on the coarsest grid level and the number of grid operations. Considering the trade-off between V-cycles and smoothing steps, there is a point for maximal performance. This can usually be determined only heuristically. In our case, it even differs for the block-asynchronous smoother and the Gauss-Seidel smoother. The reason is that the block-asynchronous iteration is not only considerably faster

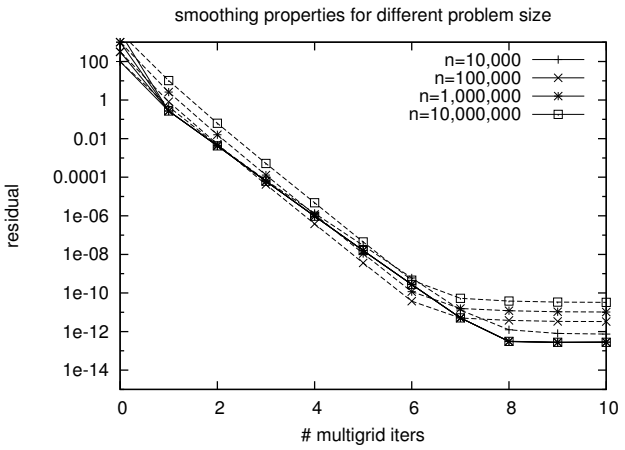


Figure 4: Two-level convergence using two smoothing step for different problem sizes. Dashed lines are block-asynchronous and solid lines are Gauss-Seidel.

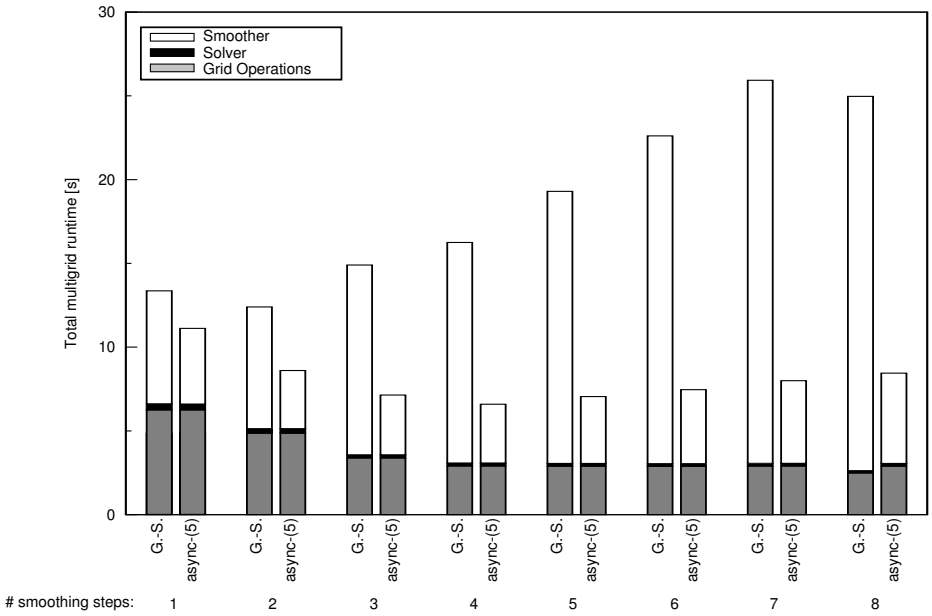


Figure 5: 10—level multigrid v-cycle runtime analysis for different numbers of smoothing steps using Gauss-Seidel and async-(5), respectively. Problem size n=$10,000,000$, condition number estimate 10^6. The async-(5) smoother includes the data transfer times from and to the GPU.

than Gauss-Seidel, but is also dominated by the data transfers between host and CPU. The component updates using a stencil for the block-asynchronous iteration come almost for free. Therefore, increasing the number of iterations does not cause a linear increase of the computation time. Hence, for large numbers of smoothing steps, the speedup factor between Gauss-Seidel and the block-asynchronous iteration smoothed multigrid rises.

Since the smoother accounts for a high percentage of the overall multigrid time, replacing the Gauss-Seidel smoother leads to considerable acceleration. While for more complex problems the ratio between smoother and overall multigrid time is often smaller, the necessity of more smoothing steps rises at the same time, making the block-asynchronous smoother even more attractive. Note that this analysis also takes the data transfer time into account. Implementing the multigrid framework on the GPU could result in even higher speedups.

4. Conclusion

We have shown that asynchronous iteration may be a suitable replacement for Gauss-Seidel or Jacobi smoothers in multigrid methods when targeting highly parallel implementations. Not only for two-grid methods but also for multigrid methods, the convergence of the multigrid solver is not affected by the chaotic behavior of the asynchronous smoother. For most test problems we were able to outperform the CPU-based Gauss-Seidel smoother by an asynchronous iteration smoother that utilized the computing power of a graphics processor. Only for small problem sizes, where the overhead of the GPU calls is crucial, was the Gauss-Seidel smoother superior. While for sequential CPU-based smoothers like Gauss-Seidel, a large number of smoothing steps directly corresponds to increased computation time, there is not this linear trade-off for GPU-based block-asynchronous iteration. Hence, choosing a larger number of smoothing steps may be reasonable and improve the overall multigrid performance. Also, adjusting the number of local iterations may have a positive impact. Since the improvement for local and global iterations depends on characteristics of the respective system, more research is necessary at this point. Another research topic may be the field of algebraic multigrid methods. There, depending on the type, asynchronous iteration can even be adapted to the structure of the multigrid method.

References

[1] H. Anzt, S. Tomov, J. Dongarra, V. Heuveline, A block-asynchronous relaxation method for graphics processing units, Tech. rep., Innovative Computing Laboratory, University of Tennessee (UT-CS-11-687, 2011).
[2] W. Hackbusch, Multigrid Methods and Applications, Wolfgang Hackbusch: Multigrid Methods and Applications, Springer, 1985, 1985.
[3] U. Trottenberg, Multigrid, Academic Press, 2000.
[4] A. H. Baker, R. D. Falgout, T. V. Kolev, U. Meier Yang, Multigrid smoothers for ultra-parallel computing, lLNL-JRNL-435315 (2011).
[5] P. Wesseling, An introduction to multigrid methods, John Wiley & Sons Inc, 1992.
[6] S. F. M. William L. Briggs, Van Emden Henson, A multigrid tutorial, SIAM, 2000.
[7] V. E. Henson, U. M. Yang, Boomeramg: a parallel algebraic multigrid solver and preconditioner, Applied Numerical Mathematics 41 (2000) 155–177.
[8] A. T. Are Magnus Bruaset (Ed.), Numerical solution of partial differential equations on parallel computers, Birkhäuser, 2006.
[9] U. M. Yang, On the use of relaxation parameters in hybrid smoothers, Numerical Linear Algebra With Applications 11 (2004) 155–172.
[10] C. T. Kelley, Iterative Methods for Linear and Nonlinear Equations, SIAM, 1995.
[11] D. Chazan, W. Miranker, Chaotic Relaxation, Linear Algebra and Its Applications 2 (7) (1969) 199–222.
[12] A. Üresin, M. Dubois, Generalized asynchronous iterations, in: CONPAR, 1986, pp. 272–278.
[13] U. Aydin, M. Dubois, Sufficient conditions for the convergence of asynchronous iterations, Parallel Computing 10 (1) (1989) 83–92.
[14] D. P. Bertsekas, J. Eckstein, Distributed asynchronous relaxation methods for linear network flow problems, Proceedings of IFAC '87.
[15] A. Frommer, D. B. Szyld, On asynchronous iterations, Journal of Computational and Applied Mathematics 123 (2000) 201–216.
[16] J. C. Strikwerda, A convergence theorem for chaotic asynchronous relaxation, Linear Algebra and its Applications 253 (1-3) (1997) 15–24.
[17] R. S. Varga, Geršgorin and his circles, Springer, 2004.
[18] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2nd Edition (August 2009).
[19] Z.-Z. Bai, V. Migallón, J. Penadés, D. B. Szyld, Block and asynchronous two-stage methods for mildly nonlinear systems, Numerische Mathematik 82 (1999) 1–20.
[20] A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Kolev, S. Martin, U. Meier Yang, Scaling algebraic multigrid solvers: On the road to exascale, Proceedings of Competence in High Performance Computing CiHPC 2010.
[21] Intel C++ Compiler Options, Intel Corporation, document Number: 307776-002US.
[22] NVIDIA Corporation, CUDA TOOLKIT 4.0 READINESS FOR CUDA APPLICATIONS, 4th Edition (March 2011).